Idea: Towards an Inverted Cloud

Raoul Strackx[1], Pieter Philippaerts[2], and Frédéric Vogels[2]

[1] iMinds-DistriNet, University of Leuven,
firstname.lastname@cs.kuleuven.be
[2] University College Leuven-Limburg,
firstname.lastname@ucll.be

Abstract. In this paper we propose the concept of an inverted cloud infrastructure. The traditional view of a cloud is turned upside down: instead of having services or infrastructure offered by a single provider, the same can be achieved by an aggregation of a multitude of *mini providers*. Even though the contribution of an individual mini provider in an inverted cloud can be limited, the combination would nevertheless be significant. We propose an architecture for an implementation of an inverted cloud infrastructure to allow mini providers to offer processor time. Security and efficiency can be achieved by building upon Intel's new SGX technology.

Keywords: protected module architectures, cloud computing, secure execution

1 Introduction

Cloud computing is a relatively young trend, having gained much traction in the past few years. Businesses and individuals alike rely on cloud computing for a variety of tasks. Cloud computing introduces a certain dynamicity to the way we work with computers. For example, a company might want to ensure availability of its systems during peak periods. One way of handling this situation consists of simply leasing additional virtual servers whenever necessary. This both reduces the total cost of ownership and allows us to deal with unexpected traffic surges robustly.

While cloud computing offers the financial advantage of sharing hardware costs among users, a significant investment is unfortunately still required from cloud providers. Meanwhile, the devices used to access cloud services are themselves underutilized. Estimates of the average utilization of computer processors vary between 5% and 15% [19].

This paper proposes a new approach, called the *inverted cloud*, which allows these idle resources to be harvested. The traditional view of a cloud is turned upside down: Instead of having services offered by a single provider, the same is achieved by an aggregation of *mini providers*. Even though the contribution of an individual mini provider in an inverted cloud can be limited, the combination would nevertheless be significant. To concretize the idea, this paper works out an architecture for an inverted cloud system to share processor time.

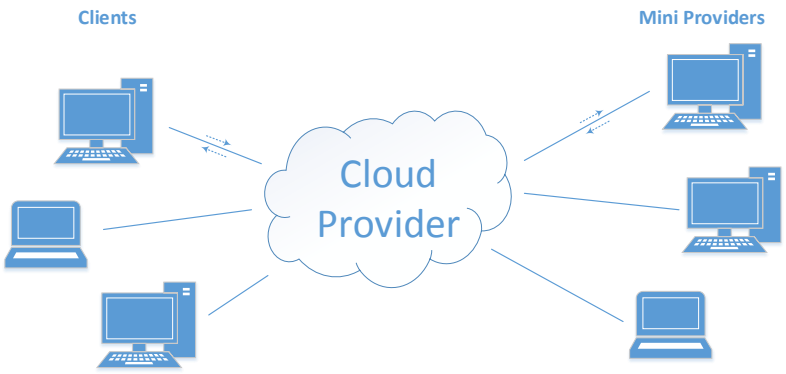

Fig. 1. An overview of the inverted cloud architecture.

The next section investigates the requirements of the proposed architecture. Section 3 works out the architecture in some detail, building upon Intel's new SGX technology to ensure security and efficiency. Section 4 discusses work related to our proposal, and finally Section 5 concludes the paper.

2 Overview and Requirements

In a standard cloud computing setting, clients send small requests to a cloud provider where they are processed and their result returned. Due to page constraints we take a more abstract approach where clients provide work packages that need to be executed. In Section 5 we discuss briefly how tasks could be split in work packages. Note that work packages are not required to be completely self-contained; They may still access other resources on the network.

In an inverted cloud setting the processing of these work packages is outsourced to mini providers. This reduces the task of a cloud provider to tracking mini providers entering and leaving the network and balancing work loads. Figure 1 displays an overview of this architecture. Clients and mini providers are only represented as two disjoint sets for clarity; There is no technical reason that a client could not also be a mini provider.

In order to be useful, the system must comply with a number of requirements. The rest of this section gives an overview of the requirements that were considered during the design of the communication and execution protocol.

Requirement 1: Secrecy of Computation Mini providers will execute work packages on behalf of clients, but they should not be able to tell *what* is being computed. The input data and the algorithm should be kept secret.

Requirement 2: Integrity of Computation Mini providers must attest that the correct, unmodified work package was executed and produced the returned result. Any manipulation of the result must be detectable by the cloud provider.

Requirement 3: Secure Execution The intermediate or output data must not be leaked; The computation must take place in a protected environment and the output should be protected.

Requirement 4: Performance The design of the system must allow for an efficient implementation.

3 Architecture

In order to execute work packages in full isolation on the resources of a mini provider, many attack vectors need to be addressed. We first introduce recent advances in protected-module architectures and sandboxing that enable such strong security guarantees, before discussing how these security primitives can be combined in a novel way to build an inverted cloud.

3.1 Protected-Module Architectures

Providing strong isolation of code and data on commodity computing devices is challenging. Operating systems have grown too complex to be able to guarantee that no defects exist that could compromise this isolation.

Security measures have been proposed to significantly raise the bar for attackers [5, 11, 18, 24, 27], but vulnerabilities (e.g., buffer overflows [13, 25]) in commodity applications and operating systems continue to be exploited on a daily basis. Any system relying on such a huge trusted computing base (TCB) cannot offer the security guarantees required to build a large inverted cloud.

Recent years many research projects have taken an alternative approach. Instead of relying on a huge TCB where the operating system provides all possible required services to applications, *protected-module architectures* (PMAs) have been proposed that provide only a minimal set of security primitives, which can be implemented in hardware [7, 9, 12, 14, 23] or by a very limited-sized hypervisor [8, 22, 26]. The exact set of primitives offered depends on the proposed security architecture, but all provide strong isolation of modules: modules are in complete control of their own memory space. Any attempt to access their memory region by code executing outside the module at *any* privilege level (including from other modules) will be blocked. Modules can only be accessed through an interface that they expose explicitly. Hence, even when the system is infested with malware, secrecy and integrity of protected modules remain guaranteed.[3] Recent work by Agten et al. [1, 2] and Patrignani et al. [16, 17] proves that high-level software properties can also be guaranteed at low-level by relying on PMA's memory protection and inserting proper checks at compile time.

PMAs avoid the snowball effect of ever-growing TCBs by using the operating system's services, while *not* trusting them completely. Strackx et al. [22] implement an example where an SSL-connection is set up between a protected

[3] In practice modules may be manipulated before protection is enabled. Such attacks are detected when the correct execution of modules is *attested* to a remote verifier or when modules attempt to access previously stored secrets.

module and a remote server. By placing application and SSL logic within protected modules, only encrypted network packets cross the modules' protection boundaries. While the operating system's services are still relied upon (e.g. for network access), these services need not be trusted. Even though kernel-level malware may modify, replay or drop network packets, confidentiality and integrity of data exchanged is guaranteed. This effectively reduces the power of an in-kernel attacker to that of a network-level attacker.

In 2013, Intel announced Software Guard eXtensions (SGX), a PMA to be implemented in their processors in the near future. Intel SGX provides even stronger security guarantees than most state-of-the-art research architectures. It not only protects modules (called "enclaves" in SGX terminology) against software-level attacks, it also guards against hardware-level attacks by ensuring that enclaves are stored unencrypted only within the processor package.

3.2 Isolating Enclaves

Protected-module architectures provide strong isolation of data stored. To enable easy integration of protected modules in legacy software, PMAs such as Intel SGX execute modules in the same address space as the rest of the application. As a result, inputs to modules do not need to be marshalled but modules can simply be provided with pointers to unprotected memory areas. Unfortunately, this also enables malicious modules to extract data stored in the same address space, or, even more worrysome, to attack the operating system.

Avonds et al. [4] and Strackx et al. [20] propose a mechanism to isolate potential attack vectors in an application (e.g., parsers) from likely attack targets (e.g., cryptographic keys). Using an approach similar to PMAs, they divide large applications in multiple compartments where each compartment can only be accessed through the interface they expose explicitly. Access to the operating system is also heavily restricted: compartments can at initialization time indicate which system calls will never be issued. Once a system call has been disabled by a compartment, it can never be re-enabled. An application that is properly compartmentalized will disable all system calls from likely attack vectors and an attacker will need to compromise multiple compartments before reaching attack targets.

For our inverted cloud infrastructure, we will use a similar sandboxing technique to protect mini providers from potentially malicious work packages. Before execution, work packages are placed in a compartment without any system call privileges. Only a very limited interface is provided to return execution results to the cloud provider.

3.3 Executing Opaque Workloads

In order to guarantee correct and safe execution, our protocol operates in two phases (see Figure 2). First, at initialization, a container enclave C is deployed on the mini provider and a public-private key pair is generated. The mini provider is now part of the inverted cloud and can receive work packages for execution. In

the second phase the private key is used by the cloud provider to send encrypted work packages to the mini provider. These work packages will be passed to the container enclave C where they are decrypted and executed in complete isolation.

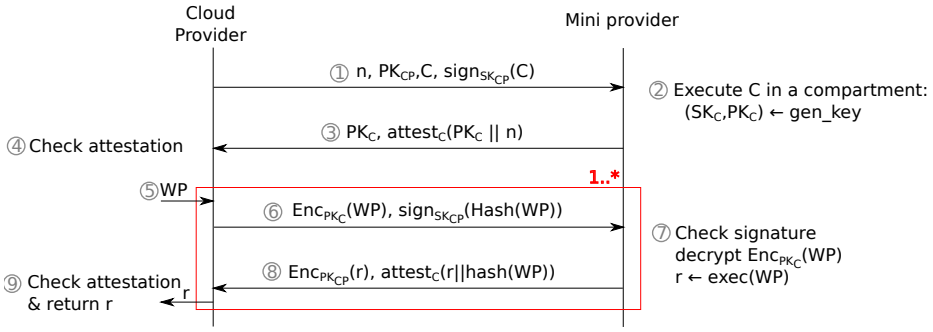

Fig. 2. The communication protocol to share work packages.

Phase 1: Initialization We assume that the mini provider supports the safe execution of potentially malicious work packages. Such support can be implemented as a stand-alone application that provides compartmentalization, or it can be integrated in an existing application such as a browser.

When the mini provider contacts the cloud provider (CP) to take part in the inverted cloud, she is provided with a (signed) container C and a nonce n (step 1 in Figure 2). After the signature of C has been verified, C is placed in a compartment. Adhering to the principle of least privilege, the compartment should only provide support to connect to the cloud provider. All system calls should be disabled.

In step 2, container C is passed to a function in the compartment where it is loaded in an enclave. Next, a private-public key pair is generated within enclave C's protection boundaries and sealed to its identity, ensuring that the cryptographic keys can be used by future instances of C. The public key is returned together with an attestation[4] guaranteeing that C was executed correctly (step 3). The enclosed nonce n ensures freshness. After the cloud provider verified the attestation (step 4) and determined that PK_C was generated by a correctly deployed and unmodified container C, the cryptographic key is stored. Work packages send to the mini provider will always be encrypted using this public key.

Phase 2: Putting Mini Providers to Work When a client sends a work package to the cloud provider, a mini provider is selected and the work package is

[4] An attestation is a log of inputs and outputs signed by a trusted entity (e.g., the mini provider's platform). To avoid linkability of attestations, Intel SGX uses a more privacy-friendly attestation scheme [3]. For clarity, we simply state what is attested.

encrypted with the mini provider's public key.[5] As this key is only accessible from container C, attackers intercepting an encrypted work package WP in step 6 cannot decrypt it. A signature is also provided to guarantee that the work package originates from the cloud provider.

In step 7, the mini provider checks the signature and decrypts the work package. Care must be taken to *erase* the container's private key before the work package is executed. Failure to do so may leak the cryptographic key to an attacker that uses the inverted cloud infrastructure to execute malicious work packages. As all work packages sent to the mini provider are encrypted with the same public key, possession of the decryption key would enable an attacker to extract sensitive information from the work packages.

When the work package finished executing, its result is encrypted with the cloud provider's public key. This result is sent back to the client, together with an attestation that the container executed correctly (step 8). The hash included in the attestation log ensures that the mini provider executed the correct work package and does not replay an old result.

The mini provider can now receive its next work package. However, enclave container C needs to be destroyed and recreated to regain access to the sealed SK_C and PK_C keys. Enabling access to the sealed cryptographic keys on disk poses a security vulnerability as it may be exploited by a malicious work package.

An alternative solution would be to load work packages in their own enclave in encrypted form. The decryption key could be passed in encrypted form to container C that provides it to the enclave after its correct set up was verified using local attestation [3]. Small bootcode at the beginning of the work package would enable decryption of the rest of the package.

4 Related Work

Outsourcing work packages to other devices is not a new idea, but related work can either not provide strong security guarantees or incurs significant overhead. The SETI@Home project, for example, distributes work loads to analyze radio signals in search for extraterrestrial life. To defend against malicious nodes returning incorrect results, the same work load is send to two different nodes. Only when both results match, the result is accepted. While effective, this approach wastes computing power and cannot guarantee confidentiality of work loads.

Recently Miller et al. [10] proposed a modification to Bitcoin. While Bitcoin clients are only required to execute otherwise useless computational workloads, they propose Permacoin, a protocol where clients are also required to store large, arbitrary volumes of data. As data can be easily confidential and integrity protected, clients can be used to store sensitive data. We propose a complementary protocol enabling (potentially malicious) mini providers to operate on that data.

Parno et al. [15] take an alternative approach to "instill greater confidence in computations outsourced to the cloud." Instead of providing a safe execution

[5] The client could encrypt the work package with the mini-provider's key when the cloud provider is not trusted, at the cost of increased communication overhead.

environment, they execute on encrypted data directly. A proof of correct execution is returned to the requester that requires less computing power than the original computation. A similar approach could be used to invert the cloud, but performance overhead to still too huge to be applied in practice.

The most related work was presented by Dunn et al. [6]. They propose the use of TPM primitives to prevent the analysis of malware. While similar, their approach cannot defend against a powerful hardware attacker. Naturally, isolation of potentially malicious work packages is also not in scope of their work.

5 Conclusion & Future Work

We have presented the concept of an *inverted cloud*. A major advantage of this approach is that cloud providers do not have to invest in resources themselves, but simply allocate resources of so-called *mini providers* to clients.

We have proposed an architecture for an inverted cloud service where clients can buy processing time. We have shown that an implementation of this idea is feasible when taking advantage of the new Intel SGX technology. The proposed architecture takes into account a number of requirements that ensure the secrecy and integrity of the computations.

In future work, we will implement and evaluate our proposed architecture. This will give us more insight in the performance of mini providers and the overhead induced by the network and communication protocol. We expect that our approach can be easily applied to solve computationally intensive work loads that can easily be split in short, parallel tasks. Applications that require long, sequential computation power may be harder to port to an inverted cloud setting, especially when mini providers may unexpectedly disconnect from the cloud network. For such work loads we look into two complementary research directions. First, mini providers may return intermediate results in the form of new work packages. Computation may then be continued by other mini providers. Second, to reduce the impact of network packet overhead and quickly disconnecting mini providers, we are looking at related work [21] to accompany work packages with digital credits. Mini providers that successfully finish execution of work packages or return intermediate results, are awarded a portion of the credits.

References

1. P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *Accepted for publication in Proceedings of the Symposium on Principles of Programming Languages (POPL'15)*, 2015.
2. P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium*, 2012.
3. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP'13*.
4. N. Avonds, R. Strackx, P. Agten, and F. Piessens. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *SecureComm'13*, 2013.

5. C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.
6. A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel. Cloaking malware with the trusted platform module. In *USENIX Conference on Security*, 2011.
7. Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.
8. J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy*, 2010.
9. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*.
10. A. Miller, E. Shi, A. Juels, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy*, May.
11. N. Nikiforakis, F. Piessens, and W. Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *DIMVA'13*.
12. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, 2013.
13. A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49), 1996.
14. E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Computer & communications security*, 2013.
15. B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (S&P'13)*, 2013.
16. M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. In *Accepted for publication in Transactions on Programming Languages and Systems*.
17. M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *APLAS'13*.
18. P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter. Code Pointer Masking: Hardening Applications against Code Injection Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.
19. M. Poniatowski. *Foundation of Green IT*. Prentice Hall, 2009.
20. R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel support for secure process compartments. In *Accepted for publication in Endorsed Transactions on Security and Safety*.
21. R. Strackx and N. Lambrigts. Idea: State-Continuous Transfer of State in Protected-Module Architectures. In *Engineering Secure Software and Systems*.
22. R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS'12*, 2012.
23. R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks*, 2010.
24. R. Strackx, Y. Younan, P. Philippaerts, and F. Piessens. Efficient and effective buffer overflow protection on ARM processors. In *WISTP'10*.
25. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *EuroSec'09*.
26. A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy*, 2013.
27. Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *ASIACCS'10*.